On the efficacy of source code optimizations for cache-based processors

Rob F. Van der Wijngaart[1] and William C. Saphir
NAS Technical Report NAS-00-014 June 2000

wijngaar@nas.nasa.gov
NAS Systems Division
NASA Ames Research Center
Mail Stop T27A-1
Moffett Field, CA 94035-1000

WCSaphir@lbl.gov
NERSC Division
Lawrence Berkeley National Laboratory
1 Cyclotron Road Mail Stop 50B-2239
Berkeley, CA 94720

Abstract

Obtaining high performance without machine-specific tuning is an important goal of scientific application programmers. Since most scientific processing is done on commodity microprocessors with hierarchical memory systems, this goal of "portable performance" can be achieved if a common set of optimization principles is effective for all such systems. It is widely believed, or at least hoped, that portable performance can be realized.

The rule of thumb for optimization on hierarchical memory systems is to maximize temporal and spatial locality of memory references by reusing data and minimizing memory access stride. We investigate the effects of a number of optimizations on the performance of three related kernels taken from a computational fluid dynamics application. Timing the kernels on a range of processors, we observe an inconsistent and often counterintuitive impact of the optimizations on performance. In particular, code variations that have a positive impact on one architecture can have a negative impact on another, and variations expected to be unimportant can produce large effects.

Moreover, we find that cache miss rates—as reported by a cache simulation tool, and confirmed by hardware counters—only partially explain the results. By contrast, the compiler-generated assembly code provides more insight by revealing the importance of processor-specific instructions and of compiler maturity, both of which strongly, and sometimes unexpectedly, influence performance.

We conclude that it is difficult to obtain performance portability on modern cache-based computers, and comment on the implications of this result.

[1]Computer Sciences Corporation, Work supported under NASA contract NAS2-14303

On the efficacy of source code optimizations for cache-based processors

Rob F. Van der Wijngaart
Computer Sciences Corporation, NASA Ames Research Center
Moffett Field, CA 94035

William C. Saphir
National Energy Research Scientific Computing Center
Berkeley, CA 94720

Abstract. Obtaining high performance without machine-specific tuning is an important goal of scientific application programmers. Since most scientific processing is done on commodity microprocessors with hierarchical memory systems, this goal of "portable performance" can be achieved if a common set of optimization principles is effective for all such systems. It is widely believed, or at least hoped, that portable performance can be realized.

The rule of thumb for optimization on hierarchical memory systems is to maximize temporal and spatial locality of memory references by reusing data and minimizing memory access stride. We investigate the effects of a number of optimizations on the performance of three related kernels taken from a computational fluid dynamics application. Timing the kernels on a range of processors, we observe an inconsistent and often counterintuitive impact of the optimizations on performance. In particular, code variations that have a positive impact on one architecture can have a negative impact on another, and variations expected to be unimportant can produce large effects.

Moreover, we find that cache miss rates—as reported by a cache simulation tool, and confirmed by hardware counters—only partially explain the results. By contrast, the compiler-generated assembly code provides more insight by revealing the importance of processor-specific instructions and of compiler maturity, both of which strongly, and sometimes unexpectedly, influence performance.

We conclude that it is difficult to obtain performance portability on modern cache-based computers, and comment on the implications of this result.

1 Introduction. Common wisdom in high performance computing is that cost-effective supercomputers will be based on commodity micro-processors with hierarchical (cache-based) memory systems. The shift away from vector supercomputers and towards cache-based distributed systems has brought about two important changes in programming paradigm. The most well-studied is that from shared to distributed memory parallelism. Less well recognized is the change in single-processor optimization strategies. Codes designed for vector machines require inner-loop independence to allow vectorization, and regular, non-power-of-two memory strides to avoid bank conflicts. Codes designed for cache-based systems require spatial and temporal locality of data usage. A cache miss is expensive (tens of cycles) and provokes a memory read of an entire cache line, not just the word being accessed. Good code uses data many times while it is in cache, and also uses neighboring data, wasting none of the cache line.

The great diversity of current processors stresses the need for *portable performance*; in addition to not requiring code changes from one platform to another (*portable syntax*), programs should run efficiently on all similar architectures, e.g., on all RISC processors with hierarchical memory systems. Syntax portability is provided by standardized languages and libraries: C, Fortran, High Performance Fortran, MPI, etc. Performance portability appears to follow from the above observations about cache-based systems, which translate into a simple set of guidelines. In scientific computing, the most cache-friendly array operations feature zero or unit stride. Moreover, loop bodies ought to be 'fat' (many assignments), so that many operations are performed on cached data. If unit stride is not possible, then one must avoid pathological strides, just as on vector computers. For cache-based systems stride issues are more complex than for vector computers, though, due to the effects of associativity and non-unit cache line size (see Bailey '95 [1]).

General processor and compiler considerations also motivate another class of guidelines. For instance, most modern micro-processors are superscalar; they complete several arithmetic operations per clock cycle, provided enough independent instructions are available. This again argues in favor of fat loop bodies. Modern processor architectures are quite complicated, however, and it is often believed that sophisticated compilers can perform machine-specific optimizations unattainable by most human application programmers, for example through loop unrolling and reordering, register optimization, software pipelining, or prefetching.

Following the above guidelines, it seems straightforward to write code that will run efficiently on any cache-based system. Practice tells us that the situation is more complicated. This paper presents observations, simulations, and some analysis of performance tuning for cache-based systems. We point out several counterintuitive results, reminding us that memory accesses are not the only factors determining performance. Moreover, our results indicate that compilers are not yet robust enough to trust them to perform all standard optimizations.

We mention a few strategies for obtaining portable performance not addressed by this paper. Vendor-optimized library routines may be useful. For instance, the Basic Linear Algebra Subroutines (BLAS) provided by many vendors, and also through the recently published self-tuning public-domain package PHiPAC [3], provide fast linear algebra kernels. While appropriate for some applications, such libraries are not appropriate for others. And even where they can be used, often substantial computation must be performed outside the library calls. Currently available source code transformation programs (e.g. the VAST and KAP preprocessors, supplemented with preprocessor directives) can ease some of the performance tuning burden, in principle. But these commercial products may not be available on all computer platforms of interest, and not all source codes are amenable to their optimization techniques. Moreover, preprocessor directives are effectively shorthand for the actual optimizations, and are thus no more portable than optimized source code itself. Hence, we will not consider preprocessors or libraries and restrict ourselves to source code optimizations carried out explicitly by the application programmer. Blocking or tiling is an effective technique for optimization on cache-based systems when many operations are performed on each data element (much reuse). The same holds for transpose methods that reduce stride in multi-dimensional problems. Blocking and transposition are not considered here, because of the limited reuse exhibited by our sample problems and by many other important

applications.

We do not attempt to explain in complete detail all performance results—although we do probe a few examples. Rather, our intent is to examine whether intuitive source-code-level optimization techniques work, and whether a standard set of such techniques can provide portable performance. This is the issue of practical importance for scientific programmers who are not experts in computer architecture and compiler design. Unlike the work by Decyk et al. [4], however, which considers only three different architectures, our study of nine current systems suggests that such a set is likely to be rather small, and to offer limited universal utility.

2 Kernel code and optimizations. The starting point for our example is the SP (Scalar Penta-diagonal) code from the NAS Parallel Benchmarks 2 (NPB 2) [2] suite. SP contains the essential elements of a computational fluid dynamics program heavily used at NASA Ames Research Center. It solves the Navier-Stokes equations using an Alternating Direction Implicit scheme. SP constitutes a stress test on the memory system of the computer, since fairly few operations per grid point are executed in any of its loops.

The most critical part of the code is the line solver, which solves systems of linear equations, one for each grid line in a 3D grid. The code boils down to solving multiple independent, banded, penta-diagonal matrix equations using Gaussian elimination. Since there are three families of grid lines in 3D space, there are also three different solver routines (*factors*). Our three code examples, named *x*-, *y*-, and *zloop*, are parts (forward elimination only) of the computational kernels of these routines. Although they have a similar structure, the memory access patterns in the three kernels are quite different, making them a good choice for comparing optimization strategies.

Each computational kernel solves a large number of independent penta-diagonal linear systems, with three right hand sides for each system. The *x*- and *yloop* fragments are shown in Appendix A. We now describe the codes in more detail. `lhs(nx,ny,nz,5)` and `rhs(nx,ny,nz,3)` are 4-index Fortran arrays of 8-byte real numbers. `nx`, `ny`, and `nz` are the dimensions of a 3D grid on which the Navier-Stokes equations are discretized. The linear systems for the *xloop* fragment are defined as follows. For fixed values `j` and `k`, `lhs(:,j,k,:)` is an `nx`×5 array containing the non-zero elements of a penta-diagonal matrix $\mathtt{A}_{\mathtt{jk}}$ of size `nx`×`nx`.[2]. Similarly, `rhs(:,j,k,:)` defines an `nx`×3 array $\mathtt{B}_{\mathtt{jk}}$ that defines three right hand sides. For each `i` and `j` we solve the independent systems $\mathtt{A}_{\mathtt{jk}}\mathtt{x}_{\mathtt{jk}} = \mathtt{B}_{\mathtt{jk}}$, where $\mathtt{x}_{\mathtt{jk}}$ is an `nx`×3 array comprising the three solution vectors. These systems are solved by Gaussian elimination. The solution vectors $\mathtt{x}_{\mathtt{jk}}$ are not stored as a separate array, but overwrite the values in `rhs`. The *xloop* fragment thus solves `ny`∗`nz` penta-diagonal linear systems that are defined along grid lines in the *x*-direction. The *yloop* and *zloop* fragments are similarly defined, except that they solve linear systems along grid lines in the *y*- and *z*-directions. In the notation above, this means that we consider a left hand side of `lhs(i,:,k,:)` for *yloop* and `lhs(i,j,:,:)` for *zloop*, and similarly for the right hand sides.

Fortran stores arrays in column-major order, so that `lhs(i,j,k,p)` is adjacent in memory to `lhs(i+1,j,k,p)`. Thus, for *xloop* successive steps of the Gaussian elimination reference elements with unit stride, while for the *yloop* and *zloop* fragments, successive steps reference

[2]We use Fortran 90 array notation to define submatrices of `lhs` and `rhs`.

elements with stride `nx` and `nx`×`ny`, respectively. Note also that the 5 non-zero elements in every row of every left hand side matrix are separated by a very large distance of `nx`∗`ny`∗`nz` elements.

The above description applies to the baseline code (no optimizations). Each subroutine is executed for four grid sizes. For simplicity we always use a cubic grid (`nx` = `ny` = `nz`). The four sizes are 16^3, 32^3, 64^3, and 80^3 points. The corresponding storage requirements are just over 0.25 MB, 2 MB, 16 MB and 32 MB, respectively. A size of `n` is henceforth used to indicate a grid of `n`×`n`×`n` points. Since pathological strides, especially powers of two, can cause recently used data to be flushed from cache [1], we pad all array grid dimensions by one unit. For example, `lhs` is actually dimensioned `lhs(nx+1,ny+1,nz+1,5)`.

In the subsequent analysis the baseline code is designated by the suffix 1. We then apply a series of *cumulative* optimizations, indicated by suffixes 2 through 5, and two additional optimizations, designated 6 and 7. In section 7 we describe an alternate, more radical optimization.

All code variations contain the same number of array references and floating point operations, but they differ in the memory access patterns, and in details of the implementations. In principle, any of the optimizations could be done automatically by the compiler, though some, especially number 4, are largely beyond current compiler technology. We examine the effect of each optimization on a wide range of modern processors. The optimizations are as follows:

1. Baseline, which is now contained in NPB 2 [2].

2. Eliminate temporaries for incremented indices, i.e. replace `i1` by `i+1`, etc. These temporaries had originally been introduced to ease programming. Removing them may enable better register allocation. "Good" compilers should be able to perform the elimination automatically, and most in fact do, as we show later. This optimization does not affect memory accesses, but we include it as a sanity check to identify compilers so bad they cannot detect constant expressions.

3. Unroll short inner loops of fixed length (`m=1,2,3`). This reduces loop overhead and register demand. Again, good compilers should be able to do this automatically, but, surprisingly, we found that several do not. The optimization does not affect memory access patterns directly, but might affect them indirectly by making it easier for compilers to interchange containing loops (see Section 6).

4. Move the last index of `rhs` and `lhs`—called the *component* index, as opposed to the *grid* indices `i`, `j`, and `k`—to the first position. This is thought to improve spatial data locality, since at each grid point all component indices of both arrays are referenced.

5. Unroll the first available loop *not* containing a recurrence (`j` for *xloop*, `i` for *yloop* and *zloop*) to a level of two. This increases the number of independent computations in the inner loop. Loop unrolling can be done automatically by a compiler, though in this case the loop bodies are rather large. This optimization does not directly affect memory access patterns.

Note the location of the assignment `fac2` (`= 1.d0/lhs(3,i,j+1,k)` in *xloop-5*). It is moved to the top of loop body, far ahead of the assignments that make use of `fac2`, to improve possibilities for optimal scheduling by the compiler.

The above optimizations all employ the *canonical* loop orderings for *x*-, *y*- and *zloop*: running index `k` for outer, `j` for middle, and `i` for the inner loop. But it is most *natural* from the application programmer's point of view to finish a whole grid line in the inner loop before moving to the next grid line. This leads to two additional code variations for *yloop* and *zloop*:

6. Use `j` and `k` as the inner loop running index for *yloop* and *zloop*, respectively, while keeping the unrollings described in optimization 5. The running index for the middle loop in each of the two loop nests is always `i`. This causes large array strides in the inner loop, which touches array elements with indices `(:,i,:,k)` for *yloop* (and similarly for *zloop*). These large strides appear bad for locality, but in the next `i`-iteration (next grid line), the inner loop touches indices `(:,i+1,:,k)`, which are adjacent. We can expect them to be in cache if the amount of data touched in the inner loop is substantially smaller than the cache size.

7. Again use the natural loop order (optimization 6), but undo unrolling optimization 5.

3 Machines. The cache-based systems in this study are mainly RISC processors: MIPS R5000, MIPS R8000, MIPS R10000, DEC Alpha EV4, DEC Alpha EV5, IBM POWER2, Sun UltraSparc I, HP PA-RISC. The one CISC architecture is the Intel PentiumPro. Almost all processors examined here are currently used in parallel platforms: MIPS R8000 in an SGI PowerChallenge, MIPS R10000 in an SGI Origin2000, DEC Alpha EV4 in the Cray T3D, DEC Alpha EV5 in the Cray T3E-900, IBM POWER2 in the SP Wide Node, Sun UltraSparc I in a workstation cluster, and HP PA-RISC in the Hewlett-Packard/Convex Exemplar SPP2000. The PentiumPro is used in several experimental PC-based clusters, including those at NASA Ames and Lawrence Berkeley National Laboratory.

Table 1: Processor specifications summary

Name	L1-cache KBytes	L2-cache KBytes	cache line Bytes	associativity	CPU MHz	Peak MFlops/s
R5000	32i+32d	0	32	2	150	300
R8000	16i+16d	4096	512	4	90	360
R10000	32i+32d	4096	128	2	195	390
EV4	8i+8d	0	32	1	150	150
EV5	8i+8d	96	32	3	450	900
POWER2	32i+256d	0	256	4	66	267
PPro	8i+8d	256	32	4	200	200
Sparc	16i+16d	512	64	1	167	334
PA-RISC	1024i+1024d	0	32	1	180	720
J90se	N.A.	N.A.	N.A.	N.A.	100	200
C90	N.A.	N.A.	N.A.	N.A.	250	1000

For comparison we also examine two vector processors, the Cray J90se and C90. These do not use caches for vector operations, but rely on memory banking and specialized hardware

to provide sufficient memory bandwidth. Machine specifics are summarized in Table 1. They reflect the modifications made to the processors to integrate them in their parallel platforms. In particular, the DEC Alpha EV4 and EV5, as used by Cray Research in the T3D and T3E, lost their off-chip L2 and L3 caches, respectively. In the T3E, the level 3 cache is replaced stream-buffer facility developed by Cray. This special hardware automatically detects adjacent cache misses, and prefetches the next cache line, since it is likely that the code is accessing data with unit stride.

One of the most important system parameters, the memory bandwidth, is not listed, because it is not directly relevant to our analysis. It is assumed that memory bandwidth is always an active constraint on processor performance of the cache-based systems. On all machines we select the highest acceptable level of optimization for the Fortran compiler (generally `-O3`). Fortran compilers supplied by the computer vendor are available for all platforms, except the PentiumPro. On the latter system the Portland Group Fortran compiler release 1.6, version 1.1 is used. Other compilers are: XL Fortran for AIX, version 4.01, on the POWER2, MIPSpro for IRIX, version 7.2, on the R10000, MIPS for IRIX, version 6.2, on the R5000 and R8000, Cray CFT77, version 6.2.3.0, on the EV4, Cray f90, version 3.0.1.1, on the EV5, WorkShop Fortran 77, version 4.2, on the UltraSparc, HP Fortran 77, version 10.30, release V1.2.1, on the Exemplar, Cray f90, version 3.0.1.0, on the C90 and J90.

4 Measured performance results. Performance figures for the kernel loop nests, in millions of floating point operations per second (Mflops/s), are presented in Appendix B.1. The numerical results are supplemented by solid disks ($\bullet$) whose magnitudes indicate the relative performance within the set of optimizations for each factor for a particular grid size. In the following paragraphs we point out a few highlights of the results contained in Appendix B.1. The main finding is that optimizations have different effects for different processors, factors, and grid sizes, so that no single choice yields portable performance. For brevity, we mention particular idiosyncrasies only once or twice, rather than every time they appear.

MIPS R5000 (Table 6). Optimization 3 (unrolling the `m`-loop) yields a marked improvement for the x-factor, and a smaller improvement the y- and z-factors. This feature, common to almost all processors, is surprising, given that it should be trivial for the compiler to unroll automatically. Optimization 4 (moving the component index) gives a substantial improvement for the y-factor and less for the x-factor, but reduces performance for the z-factor. The z-factor benefits most from optimization 6 (natural, large-stride loop ordering, as opposed to the canonical ordering).

MIPS R8000 (Table 7). Optimization 5 (unrolling the `i`-loops for the y- and z-factors) greatly deteriorates performance. We also notice the reduced processor performance for large grids that do not fit completely in cache (sizes 64 and 80), indicating insufficient bandwidth to main memory. Finally, we observe that the supposedly most cache-friendly x-factor code performs more poorly than y and z.

MIPS R10000 (Table 8). The best optimization strategy is influenced by problem size. Optimization 4 is best for small grids but is counterproductive for larger ones.

IBM POWER2 (Table 9). Moving the component index proves positive for the x- and (slightly) negative for the y- and z-factors. The best performance overall is realized by partially unrolled, natural loop nests. This leaves unexplained why the optimal x-factor

performs disproportionately well compared to the y- and z-factors.

INTEL PENTIUMPRO 200 MHZ (Table 10). The PentiumPro demonstrates great sensitivity of z-factor performance to problem size and loop ordering. The natural order is generally preferred.

DEC ALPHA EV4 (Table 11). On the DEC Alpha EV4 x-factor performance improves by unrolling the `m`-loop, whereas y- and z-factor performances deteriorate under the same code change. Moving the component index produces a dramatic increase in computational speed for the x-factor, but a much smaller improvement for the other factors. Best performance is obtained for partially unrolled `i`-loops for the y- and z-factors, but for a completely 'rolled up' `j`-loop for the x-factor.

DEC ALPHA EV5 (Table 11). The performance of this chip is influenced by the stream-buffer facility, which greatly favors unit stride access. Hence, the x factor performs best, moving the component index helps significantly, and the natural loop order with large strides does poorly. We also note that performance of the x-factor *increases* with increasing problem size, unlike all other processors (except the vector processors). Unlike a vector processor, however, improvement is realized only for long vectors of *unit* stride.

SUN ULTRASPARC I (Table 13). The UltraSparc is not very sensitive to code optimizations by the user, except for the very smallest grid size that fits entirely in the cache.

HEWLETT-PACKARD PA-RISC (Table 14). Of all the processors surveyed, the PA-RISC shows the most severe performance degradation as the problem size increases. This suggests that the PA-RISC has the greatest imbalance between memory bandwidth and processor speed of all systems investigated. We also notice a substantial drop in performance when the auxiliary variables `i1` and `i2` in the x-factor are eliminated, whereas similar code changes in the y- and z-factor have hardly any effect.

Table 2: Sum of number of instances of optimal performance

	x-factor					y-factor							z-factor						
optimization no.	1	2	3	4	5	1	2	3	4	5	6	7	1	2	3	4	5	6	7
best case tally	0	0	4	18	14	1	0	3	6	8	14	4	0	1	4	4	2	21	4

Performance results are summarized in Table 2. For each processor and each factor we count the number of grid sizes for which a particular optimization technique is superior to all others. The table lists the sum of these numbers over all processors.

There is not a single optimization strategy that gives the best performance for all grid sizes/factors for all processors. But even if we restrict the attention to one processor at a time, most still do not feature a uniform optimization strategy. Only the IBM POWER2 shows consistently best performance for one single strategy (partially unrolled, natural loop nest), although even here it is not clear whether better results could not be obtained by undoing optimizations *y/zloop-2* and *y/zloop-4*. The best overall optimization strategy is: *xloop-4*, *yloop-6*, and *zloop-6*. If we rule out partial unrolling, which is often considered impractical, then the generally most acceptable single optimization strategy is: *xloop-4*, *yloop-4*, *zloop-4*, i.e. the canonical loop ordering, and the component index as the first array index.

For comparison purposes we also show the performance results of the kernel codes on the

Cray J90 (Table 15) and C90 (Table 16). Even though there are inner loop recurrences in some cases, the Cray compilers always succeed in vectorizing the codes. With some minor exceptions, code performances improve and relative performance differences are decreased as the problem size, and hence the vector length, grows. We conjecture that the slight drop in performance of some cases on the J90 as the problem size increases from 64 to 80 is due to the fact that 80 is not a multiple of the hardware vector length, while 64 is. The results show that with little or no tuning, both machines attain approximately 33% of peak performance on the larger problems. We conclude that optimization efforts on these machines can probably be limited to increasing vector length.

5 Cache simulations. Several of the performance results in Section 4 are unexpected, as we find that minimizing inner loop stride does not consistently give best performance. In particular, the beneficial effect of the natural loop ordering for the y- and z-factors for several processors is surprising.

In this section we investigate more carefully the connection between cache miss rate and performance. The above observation about memory access implies that the simple rule of thumb of minimizing strides is not sufficient. More detailed knowledge and insight are needed to design a good optimization strategy. In this section we explore whether a simple model can improve our understanding of data locality, explain the performance results, and guide software design. We argue that while such a model can account for cache misses, the connection between cache miss rates and performance is weaker than is often believed, so that the goal of portable performance remains elusive.

We use an "intuitive" model to simulate cache behavior. It is deliberately kept simple, because its main purpose is to correct our intuition about data locality, not to provide the most detailed description of hardware performance, compiler optimizations, etc. Moreover, our interest is in determining *a priori* programming rules for portable performance on cache-based systems. Whereas detailed performance analysis is of interest in its own right (see Section 6), it is not useful for devising guidelines if too many parameters are involved that are either particular to only one or a few systems, or that are not readily quantifiable.

Other cache profiling systems, such as CProf [5] or MemSpy [6], and more general system simulation packages, such as RSIM [7] and SimOS [8], provide more accurate details regarding actual program performance. However, these tools simulate execution of assembly code, not machine-independent source code, which means they require invocation of a compiler for a specific platform. This type of simulation is useful for system design and performance tuning of an application for a particular computer, but not for portable performance prediction. Moreover, most detailed simulation tools require either many user inputs, or only support certain processor families (for instance, SimOS and CProf list support only for MIPS processors, and RSIM only for Sun SPARC V9/Solaris systems), which again limits portability and generality.

Our simple model can be applied to all cache-based systems. Its only parameters are the number of cache lines, the line size, and the associativity. We use it to determine cache behavior by transforming by hand the array references in the source code to calls to the cache simulation routines, and counting the number of cache misses as a percentage of the total number of loads.

The model has the following features:

— The highest level of cache (L1 or L2) is assumed the memory bottleneck. Its parameters are used for the simulator, and referencing a data element residing in that cache level is considered a hit. Modeling other levels as well requires knowledge about the relative bandwidth and latency between the caches, in addition to the usual parameters that characterize the lower-level cache.
— In case of set-associative caches a Least Recently Used (LRU) replacement policy of cache lines is employed. While LRU can sometimes lead to pathologically bad cache behavior, it is a reproducible and usually reasonable policy.
— Memory loads are atomic, meaning that a data item in a cache line can only be used once the whole cache line has been read from memory. We ignore sophisticated strategies such as *early restart* and *requested word first*, since these would force differentiation among cache misses.
— Effects of internal cache memory structure, such as interleaving, are not taken into account, since these would force differentiation among cache hits.
— We do not consider address translation. All cache addresses are determined directly from virtual addresses, not from physical addresses, and no lookup in page tables is required.
— We assume a separate data cache, since unified caches require vastly more information for simulation, including the assembler version of the code. Whereas this is a reasonable assumption for most L1 caches (see Table 1), most L2 caches are unified (data and instructions share the same cache). Our model effectively ignores the effects of storage of instructions in cache memory.
— Only memory loads are modeled. The structure of the computational loops in the kernel code is such that stores are always done to memory locations that are already in cache, which means that no cache lines need to be flushed to accommodate write operations. We ignore the effects of writing through the cache to main memory.
— Loads are ordered canonically, meaning that calls to the cache simulation load routines are inserted in the order in which array references occur in the source code. This precludes out-of-order execution, and also ignores other possible compiler optimizations that are impossible to anticipate without inspecting the assembler code.
— The number of registers is assumed large enough to accommodate all scalars occurring in the kernel code, so that their storage does not compete for space in the data cache.

Note that our cache model is similar to that used in the loop nest interface of the Cache Visualization Tool [10], except that CVT models only direct-mapped caches. Relative array base addresses are fixed by placing `rhs` and `lhs` in a common block, separated by a spacer of known size.

We report here the results of simulations of a subset of the optimizations: 3, 4, and 7. This subset encompasses the major differences in memory access patterns. Optimizations 1, 2, and 3 have identical memory reference patterns (according to our simulation rules and intuition); we only present results for 3. Optimizations 5 and 6 correspond to 4 and 7, respectively, except for the partial loop unrolling. Our simulations show that the latter has a negligible effect on array reference patterns and cache misses, so we only present figures for 4 and 7.

Results of the simulations are contained in Appendix C. Each table shows the *percentage* of array references (loads) that cause cache misses for the optimizations described in Section

2 and presented in Appendix A.1. Symbols are applied in the same fashion as in the tables in Appendix B.1: the larger the ring (⊙), the fewer the cache misses.

We expect a smaller percentage of cache misses to result in faster code. If an optimization results in fewer cache misses and higher performance than another, we say there is a positive correlation between simulation and measured performance. In the following we focus on large changes in cache miss rate or performance, and, as before, limit our discussion to the more striking examples. Our overall finding is that sometimes cache miss rates do correlate positively with performance, and sometimes they do not. In other words, even a sharpened attention to cache behavior does not result in achieving portable performance.

MIPS R5000 (Table 28). Cache misses and performance are relatively well correlated. For instance, performance increases and cache misses decrease for *zloop-4,3,7* and *yloop-3,7,4*. Cache behavior also explains the increase in performance with larger grid sizes for *xloop*, and the mixed results with larger sizes for *yloop* and decreased performance with larger sizes for *zloop* (except for *zloop-7*). On the other hand, cache behavior does not explain everything, such as the dramatic increase in performance from *yloop-3* to *yloop-4* for the smaller two problems.
We note that the number of cache misses for the z-factor is slashed in three—somewhat unexpectedly—by using the natural loop ordering. This is reflected in a better performance.

MIPS R8000 (Table 29). This processor features a very small miss rate (less than 1%), due to the large size of the L2 cache. The large performance differences between the factors, particularly for the smaller grid sizes, is not explained by differences in L2 cache misses. A more likely explanation is that for problems that fit inside L2 cache, it is L1 cache misses that determine performance.

MIPS R10000 (Table 30). For small grid sizes we observe no correlation, and in fact for a grid size of 32 there is a negative correlation. For the larger sizes, modest changes in performance are not reflected in the cache miss rate.

IBM POWER2 (Table 31). Generally speaking, variations in cache misses and performance are relatively small, though the two are not well correlated. Where there is the largest variation in cache miss rate (*zloop*), performance correlates negatively. Conversely, the virtually uniform distribution of cache miss rates for *xloop* is accompanied by the largest variations in measured performance.

Intel PentiumPro 200 MHz (Table 32). For the largest problem size correlation appears quite good—fairly uniform performance for the x- and y-factors, and widely varying performance with high correlation for the z-factor. For the smallest case, again we see uniformity in the x- and y-factors, but note that cache misses are uniform for the z-factor, while performance shows the same dramatic swings as for the large grid size. This casts doubt upon the significance of the z-factor correlation for the largest size.

DEC Alpha EV4 (Table 33). Cache misses fail to predict the large performance variation within the x-factor, and are only moderately correlated elsewhere.

DEC Alpha EV5 (Table 34). A reduction in cache misses of up to a factor of 3 for the larger grids of the z-factor by switching to the natural loop ordering has no or no positive effect on the performance. All factors show widely varying performance figures for virtually identical numbers of cache misses (grid size 16).

Sun UltraSparc I (Table 35). Performance of the UltraSparc for small grids (16) varies significantly, especially for the x-factor, although the problem fits entirely in the cache, and

no misses occur (see Table 35). For larger grids the number of cache misses varies significantly for the z-factor, but this is not correlated with performance.

HEWLETT-PACKARD PA-RISC (Table 36) . The most salient feature of the simulated cache behavior of the PA-RISC is the strong negative correlation between measured performance and number of cache misses of the z-factor for large grids.

In summary, the simulated cache behavior for the processors studied correlates poorly with the actually observed performance, confirming the conclusion from the previous section that simple rules taking into account a hierarchical memory structure are not sufficient to ensure portable performance. Most strikingly, the smaller grid sizes fit entirely in the L2 caches of some processors, but observed performances for the various optimizations differ substantially. Less dramatic, but equally vexing, is the negative correlation between measured performance and calculated cache misses for the y-factors on the PentiumPro, DEC Alpha, Sun UltraSparc and IBM POWER2. Finally, we also observe that measured performances of optimizations 1, 2, and 3 for all factors vary significantly, although the array reference patterns for these code variations are identical.

It can be argued that these discrepancies are due to the limitations of the simulator—perhaps better termed a *data locality estimator*—and that better correlations can be obtained by incorporating more detailed characteristics of the particular machines. But we are interested in producing *portable* code, and introducing even more information into the program construction will make this task virtually impossible. In addition, the validity of the cache simulator is corroborated by program run-time statistics obtained from hardware performance counters on the IBM POWER2 and the MIPS R10000. Although on both machines the numbers of simulated cache misses overpredict the actually measured numbers, the *correlation* between the two is higher than 95%.

We also observe that there is significant—and nontrivial—dependence of the processor performance on the problem size, which is generally not known at compile time. We do note that on several processors the unexpected beneficial effect of using the natural loop ordering for the z-factor is borne out by the simulations. However, the data locality of the simple pieces of code investigated in this paper is generally a rather complex function of loop organization, problem size, and cache structure, even if only very few parameters are used to describe the cache.

6 Performance details. Clearly, it is impossible to explain the performance of the kernel codes by only examining the source text, even when our intuition about data locality has been improved through the use of a cache simulator. In order to understand better why this is, and to gain appreciation of the factors that do govern performance, we examine the generated assembly code for some of the processors in more detail. Through this detailed analysis we are able to explain almost all significant variations in performance. While it is gratifying to know that much can be explained, the reliance on compiler- and processor-specific details means that the goal of portable performance cannot be achieved.

The selected processors are the MIPS R8000, IBM POWER2, and DEC Alpha EV4. Each has 32 integer and 32 floating-point general-purpose registers, plus a small number of special-purpose registers. The MIPS and IBM processors feature a combined floating-point multiply/add (*madd*) instruction, which is missing from the DEC Alpha instruction set. All

kernel code assignments written as a combination of an addition (or subtraction) and a multiplication are correctly recognized as madds by the MIPS and IBM compilers. In addition, the IBM processor is capable of loading or storing two double-precision numbers (occupying four 32-bit words, hence the names *quad* load and *quad* store) in a single instruction. The memory buses of the MIPS and DEC Alpha processors, in contrast, are only 64 bits wide, and do not accommodate quad loads or stores. Finally, the IBM instruction set features an address mode (called the *update* mode) that allows simultaneous update and use of the contents of a register. This mode, which saves on the number of register manipulations, is absent from the MIPS and DEC Alpha instruction sets.

In Tables 3, 4, and 5, we summarize numerically the characteristics of the generated assembly codes. The quantities *load* and *store* refer to the number of memory operations carried out for each point of the grid. For the POWER2 we indicate, in parentheses, how many of the loads and stores are quad operations. For the EV4 we show, also in parentheses, how many of the loads and stores are between registers and stack variables (as opposed to array variables in main memory). Since stack variables are only few and, presumably, remain in cache for the duration of the kernel code execution, such loads and stores are less costly than those of array variables. The R8000 and POWER2 inner loops have no stack loads or stores. *Instr* signifies the total number of assembly instructions per grid point, and *sep* indicates the separation, or delay, between issuing the computationally expensive division instruction and using the result. A sep(aration) of 1 means that the instruction immediately following the division makes use of the result. This delay can be important when the processor allows other instructions following the division to be carried out before the division completes. Note that we do not take into account that instructions may have different lengths (and, consequently, different costs), depending on their address mode. For the determination of the above four numerical parameters we ignore any operations that are not in the inner loop. We note that none of the compilers investigated recognizes the possibility of postponing writing array elements back to memory (storing) until operations on them have been completed (see Section 7).

MIPS R8000. The MIPS compiler uses loop replication and pipelining as the most important vehicle for code optimization. This technique has the potential to reduce the number of loads (through reuse among different iterations), and to increase the division separation (through reordering of statements), at the cost of pipeline overhead and the danger of register spill due to the replication. Only innermost loops are pipelined, and only if they are of sufficient depth to justify the overhead. This disqualifies the small `m`-loops, which need to be unrolled before pipelining can proceed. The MIPS compiler automatically unrolls the first `m`-loop of the loop body (i.e. `rhs(i,j,k,m) = fac1*rhs(i,j,k,m), m=1,2,3`), but leaves the other two unchanged. The recursion in `i` for the x-factor prevents the inner loop from being split without loop reordering, and no pipelining is performed at all. This explains the relatively poor performance of *xloop-1,2*. Once all `m`-loops are unrolled by hand (*xloop-3*), the inner loop is replicated three times (in the `i`-direction) and fully pipelined, leading to a greatly reduced number of loads and instructions, and a substantial increase of the division separation. The resultant performance improvement is roughly a factor of two for all grid sizes. Changing the position of the component index (*xloop-4*) has hardly any effect on the structure of the compiled code. Performance also remains effectively the same. Unrolling the `i`-loop by a factor of two (*xloop-5*) renders the loop body too large for the

Table 3: MIPS R8000

	load	store	instr	sep
x-1	32	15	100	2
x-2	32	15	100	2
x-3	13	15	46	30
x-4	13	15	47	31
x-5	32	15	66	33
y-1	32	15	79	40
y-2	32	15	79	40
y-3	18	15	51	60
y-4	17	15	51	61
y-5	32	15	68	15
y-6	32	15	68	15
y-7	18	15	51	61
z-1	30	15	93	3
z-2	30	15	92	3
z-3	17	15	50	59
z-4	18	15	52	62
z-5	35	15	68	15
z-6	35	15	68	15
z-7	18	15	52	62

Table 4: IBM POWER2

	load	store	instr	sep
x-1	$30(2^a)$	15	72	22
x-2	$30(2^a)$	15	70	18
x-3	$27(5^a)$	15	59	24
x-4	$13(7^a)$	$8(7^a)$	37	8
x-5	$13(7^a)$	$8(7^a)$	37	19
y-1	32	15	74	17
y-2	32	15	76	13
y-3	32	15	64	21
y-4	$22(6^a)$	$13(2^a)$	52	12
y-5	$22(6^a)$	$13(2^a)$	51	29
y-6	$17(3^a)$	$13(2^a)$	50	17
y-7	$20(6^a)$	$13(2^a)$	50	5
z-1	32	15	74	17
z-2	32	15	76	13
z-3	32	15	64	22
z-4	$22(6^a)$	$13(2^a)$	52	12
z-5	$22(6^a)$	$13(2^a)$	51	29
z-6	$20(6^a)$	$13(2^a)$	50	34
z-7	$20(6^a)$	$13(2^a)$	50	5

[a]quad load/store

Table 5: DEC Alpha EV4

	load	store	instr	sep
x-1	$37(4^a)$	$21(6^a)$	128	6
x-2	$37(4^a)$	$21(6^a)$	128	6
x-3	27	15	80	13
x-4	17	15	62	11
x-5	17	15	62	33
y-1	30	15	126	3
y-2	30	15	126	3
y-3	30	15	96	19
y-4	24	15	75	4
y-5	24	15	70	31
y-6	23	15	69	29
y-7	23	15	74	4
z-1	36	15	126	3
z-2	36	15	126	3
z-3	$33(3^a)$	15	96	19
z-4	24	15	75	4
z-5	24	15	70	31
z-6	23	15	74	29
z-7	23	15	73	4

[a]stack load/store

compiler to optimize, and no pipelining is done. The number of loads goes up again, but the absence of `m`-loop overhead and the programmer-induced increased division separation keep the performance from deteriorating precipitously.

In case of the *y*-factor, the compiler again fails to unroll the second and third `m`-loops (*yloop-1,2*). But because the inner grid loop runs over points in the `i`-direction and the recursion is in the `j`-direction, the inner loop can be split into four independent loops, each of which is optimized separately. The two that only contain an `m`-loop are subjected to loop inversion, so that the innermost loops have `i` as the running variable. This allows these loops to be replicated (in the `i`-direction) and pipelined, which increases the division separation. Moreover, `m`-loop overhead is moved out of the inner loop, leading to a noticeable reduction of the number of operations compared to *xloop-1,2*. As a consequence, *yloop-1,2* perform significantly better than their *x*-factor counterparts. The two inner loops that do not contain `m`-loops are replicated and pipelined directly. This yields no additional gain, since splitting the `i`-loop prevents reuse among and within iterations; the total number of loads is the same as for the *x*-factor. Unrolling the `m`-loops by hand (*yloop-3*) enables the compiler to replicate and pipeline the entire inner loop directly, further increasing the division separation. The number of loads (and hence the number of instructions) is sharply decreased because of reuse of `lhs` array elements within the single loop body. Notice that the division separation is much larger than for the corresponding *x*-factor code. This is because the result of the division is now not needed by the next iteration of the inner loop. As a consequence, performance is improved substantially. Changing the position of the component index (*yloop-4*) leaves the structure of the compiled code and the performance again virtually unchanged. When the inner loop is unrolled by hand by a factor of 2 (*yloop-5*), the loop body is again too complex for the compiler to optimize, and no pipelining is done at all; strangely, even though the

expensive divisions are scheduled together at the top of the inner loop by the programmer, creating the opportunity to increase division separation for the second iteration, the compiler moves the second division down to the start of its 'own' iteration. This reordering limits the division separation and degrades performance to a level well below that of *xloop-5*. Inverting the loops to reflect the natural loop order (*yloop-6,7*) is undone by the compiler, which recasts these kernel codes to exhibit the canonical loop order. Consequently, structure and performance of these compiled codes are identical to those of *yloop-5,4*, respectively.

The same transformations applied to *yloop-1,2* could have been used to optimize *zloop-1,2*, but are not found by the compiler. Instead, the `j`-loop is broken into only three independent parts. The first contains the expensive division as well as a non-expanded `m`-loop. This combination of instructions inhibits loop inversion and pipelining. Although the other two loops are again replicated and fully pipelined, the poor optimization of the first loop leads to a relatively large number of instructions and a very small division separation, thus explaining the bad performance of these kernel codes. Unrolling the `m`-loops by hand (*zloop-3*) enables the full range of optimizations applied also to *yloop-3*, with concomitant performance improvement. As before, moving the component index (*zloop-4*) affects neither the structure of the compiled code, nor its performance. As in *yloop-5*, the unrolled loop body of *zloop-5* is too large to pipeline, and again the compiler migrates the division operation, to the detriment of the division separation. When the natural loop order is adopted, the compiler again attempts to reduce stride by recasting the loop nest, but instead of reverting to the canonical loop order (i.e. `k` for outer, `j` for middle, and `i` for innner loop running indices), the resulting loop nest is `j-k-i`. Apparently, this does not affect performance much. Performance of the fat, non-pipelined *zloop-6* code is on a par with that of *yloop-6*. *Zloop-7* and *yloop-7* compare similarly.

IBM POWER2. Unlike the Cray (see below) and MIPS compilers, the IBM compiler always recognizes and expands the short `m`-loops to avoid unnecessary branch instructions. It also pipelines each inner loop to reduce memory operations and increase division separation. Once the `m`-loops are unrolled by hand (*x,y,zloop-3*), the compiler has an easier job recognizing access and dependency patterns, and for all three factors the number of instructions decreases while the division separation goes up. Performance improvement is greatest for the x-factor, since now the compiler also recognizes the possibility of utilizing array elements adjacent in memory, and reduces the number of load operations by issuing some quad loads. Increased locality of memory reference is obtained by moving the component index (*x,y,zloop-4*), which enables quad loads as well as quad stores for all factors, thus reducing the number of memory operations as well as the total instruction count. However, although the code is still fully pipelined, for some reason the compiler now fails to unroll the inner loop (so far, each was unrolled automatically by a factor of two in the `i`-direction), which limits the division separation. As a result, the net performance improves only noticeably for *xloop-4*, not *y,zloop-4*, because the x-factor experiences the sharpest reduction of the number of memory operations due to quad load/stores. Unrolling the inner loop by hand by a factor of two (*x,y,zloop-5*) restores the division separation for all three factors while leaving other parameters virtually unchanged, and performance improves commensurately.

Reverting to the natural loop order reduces the number of loads for the y-factor, but the division separation also decreases, so it is not clear why performance of *yloop-6* improves. For the z-factor the number of loads as well as the division separation are affected favorably by

the natural loop order, and the number of cache misses is reduced by a factor of three for the larger grid sizes, which explains the improved performance of *zloop-6*. Undoing the partial unrolling of the `i`-loop leads to a substantial reduction of the division separation, and, for the y-factor, also to an increase in the number of loads; as expected, the performance of *y,zloop-7* deteriorates. Overall, performance of the x-factor on the POWER2 is significantly better than of the y- and z-factors. Data locality alone does not provide sufficient explanation. What matters also is the *type* of locality, namely adjacency.

DEC Alpha EV4. Unlike the MIPS and IBM compilers, the Cray compiler never replicates or pipelines the kernel code loops automatically, and hence has fairly little control over division separation and number of memory operations. *Xloop-1,2*, which yield virtually identical assembly code, fare poorest. Like MIPS, the Cray compiler expands the first `m`-loop, but leaves the others untouched. Because the inner grid loop contains a recurrence, it cannot be split, and is left virtually unchanged by the compiler. Many register spills occur, and even the loop constant `3`, used for the short `m`-loops, is read from the stack several times. In addition, there are redundant loads of unchanged values. Apparently, the compiler made only very minor attempts at optimization, resulting in large numbers of instructions and memory operations, small division separation, and bad performance. When the `m`-loops are unrolled by hand (*xloop-3*), loop overhead is cut, also leaving more registers available, so fewer spills occur. Some reordering of statements takes place, and the compiler issues loads further ahead, which leaves more time for data to be fetched and also increases the division separation. Moving the component index (*xloop-4*), however, yields a much bigger performance improvement, which is due to the large reduction of the number of loads and the total number of instructions. The reason for this is that address calculations are now a lot simpler. When the component index is the last array index, 8 integer registers are used to keep track of the 5 elements of `lhs` and 3 of `rhs` at each grid point, since these are separated by large strides, the sizes of which are unknown at compile time. These registers need to be updated every iteration (recall that the EV4 lacks an update address mode), resulting in significant overhead. Moreover, due to the large number of registers used, bookkeeping is relatively complicated, and the compiler issues several redundant loads. By contrast, when the component index is first, only two integer registers are required to keep track of all local elements of `lhs` and `rhs`. All other addresses are obtained using small (24 or 40 bytes), fixed-size offsets. Moreover, due to the simple structure of the resulting code, the compiler is able to eliminate all redundant loads, which leads to a very streamlined, short program. Partially unrolling the `j`-loop (*xloop-5*) leaves that structure intact and increases the separation division, but the number of cache misses also increases, and performance goes down.

Both *yloop-1,2* and *zloop1,2* permit splitting the inner `i`-loop, which the compiler does correctly. It also employs loop inversion, so that the `m`-loops are no longer in the inner loop, and loop overhead is cut. The smaller number of instructions and substantially reduced number of loads makes the y-factor perform more than 50% better than the x-factor. The z-factor sheds fewer loads, and its performance improves less, compared to the x-factor. When the `m`-loop is expanded by hand (*y,zloop-3*), the number of memory operations, total number of instructions, and division separation all improve or stay the same, but performance degrades nonetheless. True performance improvement is brought about by the change of position of the component index (*y,zloop-4*), which results in a sizeable reduction of number

of loads and total instruction count. Despite the fact that the cache miss rate goes up and the division separation goes down, performance improves. Apparently, either divisions are relatively cheap on the EV4, or delayed instruction completion is not possible, as partially unrolling the `i`-loop (*y,zloop-5*) keeps the structure and properties of the assembly code the same, save a substantial increase of the division separation, but performance declines somewhat. Adopting the natural loop order (*y,zloop-6*) cuts the cache miss rate in half, improving performance substantially. Finally, undoing the partial unrolling (*y,zloop-7*) makes the cache miss rate grow slightly and decreases the division separation, neither of which appears significant enough to explain the fairly steep drop in performance.

7 Alternate optimizations. The optimizations of the kernel code presented in Section 2 affect mainly loop structure, but leave the number of array references unchanged. Here we present an alternate optimization strategy, due to Taft [9], that aims to reduce the number of array references. It is based on the following observations. In every iteration three successive rows of the pentadiagonal matrix and the corresponding three triplets of right hand side values are read and modified. Several of the matrix elements are accessed multiple times. When the inner loop corresponds to the line solve direction (natural loop order), the first two of the three rows accessed in the next iteration coincide with the current triplet. The number of array references can therefore be reduced by storing triplets of rows in a 'moving window' of scalar variables. After the window has been initialized at the beginning of the grid line, all that is required to move the window between iterations is to read one new matrix row from memory and store back the elements of the first row that have changed.

If enough registers are available, all operations on the window of scalars can take place entirely in registers. If not, the spilled scalars will reside in fixed positions on the stack, and all window data will likely remain in cache; no new reads from main memory are necessary after each new matrix row has been stored in scalar variables. Good compilers already recognize some reuse of array elements within a single iteration and store them in registers, but fairly substantial transformations are required to take full advantage of reuse across several iterations. This is evidenced by the complexity of source code employing the scalar window (Appendix A.2).

Performance results (Mflops/s) are tabulated in Appendix B.2. Since the scalar window applies only to the natural loop ordering, optimizations 1–5 for *yloop* and *zloop* are not displayed. We also omit optimizations *xloop-1,2*, so that the entire inner loop can be written in scalar operations (no `m`-index loops). We observe several appreciable performance improvements (up to 20%) over the kernel codes, especially for the x-factor on the PA-RISC (large grids) and the R8000, the y-factor on the PA-RISC (large grids), the UltraSparc (small grids) and the POWER2, and the z-factor on the PA-RISC (medium grids), the UltraSparc (small grids), and the POWER2 and EV5. But there are also dramatic performance degradations (up to 50%) for the x-factor on the EV4, the y-factor on the R8000 and EV5, and the z-factor on the R8000 and R10000 (large grids). Consequently, this code variation again cannot be recommended as a portable optimization technique for cache-based systems. As expected, the vector machines do not benefit at all from the scalar window optimizations.

8 Conclusions. We have studied the behavior of variations of pieces of scientific computing software on a wide range of current cache-based processors. Seemingly reasonable source

code optimizations often did not yield higher speed, even when cache simulations indicated they would. Moreover, the same "optimization" often had very different consequences for performance on different processors.

Our goal was to find out whether it is feasible to achieve portable performance on computers with hierarchical memory systems. Our conclusion is that it is not. We found instead that many performance fluctuations could be understood only by examining assembly code in detail, and ultimately these fluctuations were related to idiosyncrasies of architectures and compilers.

Tuning codes for high performance on commercial cache-based processors available today is a process that requires so much knowledge and information about the entire configuration of user program, problem parameters, system software, and hardware organization, that it is in general difficult or impossible to write good, portable cache code for general scientific applications. By this we mean that it is not possible to tell whether a program will perform well by studying the source code and a few high-level parameters of the system under consideration. This contrasts sharply with our experiences on vector processors, where the well-understood effects of vectorizability, regular stride, and long vector length virtually completely govern performance. We note also that compilers are often not as sophisticated and mature as is commonly believed. Many still fail to perform automatically even trivial optimizations.

Generic optimization strategies designed to take advantage of cache by increasing data locality do not alone yield high performance. Other factors, such as software pipelining, compiler structure and maturity, (relative) instruction cost, number of registers, and special memory and address instructions, need to be considered as well when designing efficient codes, but this will hamper portability of codes between architectures.

References

[1] D.H. Bailey, "Unfavorable Strides in Cache Memory Systems," *Scientific Programming*, vol. 4 pp. 53–58, 1995

[2] D.H. Bailey, T. Harris, W.C. Saphir, R.F. Van der Wijngaart, A. Woo, M. Yarrow, "The NAS Parallel Benchmarks 2.0," *NAS Technical Report NAS-95-020*, NASA Ames Research Center, Moffett Field, CA, 1995

[3] J. Bilmes, K. Asanović, C-W. Chin, J. Demmel, "Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology," *Proc.* 11^{th} *ACM Intl. Conf. on Supercomputing*, Vienna, Austria, July 1997

[4] V.K. Decyk, S.R. Karmesin, A. de Boer, P.C. Liewer, "Optimization of particle-in-cell codes on reduced instruction set computer processors," *Computers in Physics*, vol. 10, no. 3, 1996

[5] A.R. Lebeck, D.A. Wood, "Cache profiling and the SPEC benchmarks: a case study," *IEEE Computer*, Vol. 27, No. 10, October 1994

[6] M. Martonosi, "Analyzing and tuning memory performance in sequential and parallel programs," *Ph.D. Thesis*, Computer Science Dept., Stanford Univ., Stanford, CA, December 1993

[7] V.S. Pai, P. Ranganathan, S.V. Adve, "RSIM Reference Manual, Version 1.0," *Technical Report 9705*, Dept. of Electrical and Computer Eng., Rice Univ., Houston, TX, August 1997

[8] M. Rosenblum, E. Bugnion, S. Devine, S.A. Herrod, "Using the SimOS machine simulator to study complex computer systems," *ACM Trans. on Modeling and Computer Simulation*, Vol. 7, No. 1, January 1997

[9] J. Taft, "Private communication," April 1997

[10] E. Van der Deijl, G. Kanbier, O. Temam, E.D. Granston, "A cache visualization tool," *IEEE Computer*, Vol. 30, No. 7, July 1997

Appendix A: Source codes.

Source listings of the SP solver fragments used in the performance tests; only the forward elimination parts of the penta-diagonal line solvers are used. For each of the three factors to be inverted in the ADI scheme (corresponding to the x-, y-, and z-directions), the same optimizations are performed for corresponding suffixes. For example, *xloop-3*, *yloop-3* and *zloop-3* all unroll inner loops of length three. *Italic typeface* is used to indicate which parts of the code fragments are affected by the current optimization.

A.1: Kernel codes

All code fragments use standard (indexed) array references throughout. *xloop-1*, *yloop-1* and *zloop-1* are the actual loops used in NPB 2. Here we only show *xloop-1* through *xloop-5*, *yloop-6*, and *yloop-7*. The other code fragments are easily inferred.

xloop-1: NPB 2.2 code

```
  do 2  k=1,nz
   do 2  j=1,ny
    do 2  i=1,nx-2
     i1=i+1
     i2=i+2
     fac1        =1.d0/lhs(i,j,k,3)
     lhs(i,j,k,4)=fac1*lhs(i,j,k,4)
     lhs(i,j,k,5)=fac1*lhs(i,j,k,5)
     do 5  m=1,3
       rhs(i,j,k,m)=fac1*rhs(i,j,k,m)
5    continue
     lhs(i1,j,k,3)=lhs(i1,j,k,3)-
 >                 lhs(i1,j,k,2)*lhs(i,j,k,4)
     lhs(i1,j,k,4)=lhs(i1,j,k,4)-
 >                 lhs(i1,j,k,2)*lhs(i,j,k,5)
     do 8  m=1,3
       rhs(i1,j,k,m)=rhs(i1,j,k,m)-
 >                lhs(i1,j,k,2)*rhs(i,j,k,m)
8    continue
     lhs(i2,j,k,2)=lhs(i2,j,k,2)-
 >                lhs(i2,j,k,1)*lhs(i,j,k,4)
     lhs(i2,j,k,3)=lhs(i2,j,k,3)-
 >                lhs(i2,j,k,1)*lhs(i,j,k,5)
     do 2 m=1,3
      rhs(i2,j,k,m)=rhs(i2,j,k,m)-
 >               lhs(i2,j,k,1)*rhs(i,j,k,m)
 2 continue
```

xloop-2: remove auxiliary variables for incremented indices

```
  do 2  k=1,nz
   do 2  j=1,ny
    do 2  i=1,nx-2
     fac1        =1.d0/lhs(i,j,k,3)
     lhs(i,j,k,4)=fac1*lhs(i,j,k,4)
     lhs(i,j,k,5)=fac1*lhs(i,j,k,5)
     do 5  m=1,3
      rhs(i,j,k,m)=fac1*rhs(i,j,k,m)
5    continue
     lhs(i+1,j,k,3)=lhs(i+1,j,k,3)-
 >            lhs(i+1,j,k,2)*lhs(i,j,k,4)
     lhs(i+1,j,k,4)=lhs(i+1,j,k,4)-
 >            lhs(i+1,j,k,2)*lhs(i,j,k,5)
     ....
```

xloop-3: unroll small inner loops of length 3

```
  do 2  k=1,nz
   do 2  j=1,ny
    do 2  i=1,nx-2
     fac1        =1.d0/lhs(i,j,k,3)
     lhs(i,j,k,4)=fac1*lhs(i,j,k,4)
     lhs(i,j,k,5)=fac1*lhs(i,j,k,5)
     rhs(i,j,k,1)=fac1*rhs(i,j,k,1)
     rhs(i,j,k,2)=fac1*rhs(i,j,k,2)
     rhs(i,j,k,3)=fac1*rhs(i,j,k,3)
     lhs(i+1,j,k,3)=lhs(i+1,j,k,3)-
 >            lhs(i+1,j,k,2)*lhs(i,j,k,4)
     lhs(i+1,j,k,4)=lhs(i+1,j,k,4)-
 >            lhs(i+1,j,k,2)*lhs(i,j,k,5)
     rhs(i+1,j,k,1)=rhs(i+1,j,k,1)-
 ¿          lhs(i+1,j,k,2)*rhs(i,j,k,1)
     rhs(i+1,j,k,2)=rhs(i+1,j,k,2)-
 ¿          lhs(i+1,j,k,2)*rhs(i,j,k,2)
     rhs(i+1,j,k,3)=rhs(i+1,j,k,3)-
 ¿          lhs(i+1,j,k,2)*rhs(i,j,k,3)
     ....
```

xloop-4: move component index of rhs/lhs to front

```
  do 2  k=1,nz
   do 2  j=1,ny
    do 2  i=1,nx-2
     fac1          =1.d0/lhs(3,i,j,k)
     lhs(4,i,j,k)=fac1*lhs(4,i,j,k)
```

```
      lhs(5,i,j,k)=fac1*lhs(5,i,j,k)
      rhs(1,i,j,k)=fac1*rhs(1,i,j,k)
      rhs(2,i,j,k)=fac1*rhs(2,i,j,k)
      ....
```

xloop-5: unroll j-loop to a level of 2

```
  do 2  k=1,nz
   do 2  j=1,ny, 2
    do 2  i=1,nx-2
     fac1        =1.d0/lhs(3,i,j,k)
     fac2        =1.d0/lhs(3,i,j+1,k)
     lhs(4,i,j,k)=fac1*lhs(4,i,j,k)
     lhs(5,i,j,k)=fac1*lhs(5,i,j,k)
     rhs(1,i,j,k)=fac1*rhs(1,i,j,k)
     rhs(2,i,j,k)=fac1*rhs(2,i,j,k)
     ....
     lhs(4,i,j+1,k)=fac2*lhs(4,i,j+1,k)
     lhs(5,i,j+1,k)=fac2*lhs(5,i,j+1,k)
     rhs(1,i,j+1,k)=fac2*rhs(1,i,j+1,k)
     rhs(2,i,j+1,k)=fac2*rhs(2,i,j+1,k)
     ....
```

yloop-6: natural loop order, i-loop unrolled to level 2

```
  do 2  k=1,nz
   do 2  i=1,nx,2
    do 2  j=1,ny-2
     fac1        =1.d0/lhs(3,i,j,k)
     fac2        =1.d0/lhs(3,i+1,j,k)
     lhs(4,i,j,k)=fac1*lhs(4,i,j,k)
     lhs(5,i,j,k)=fac1*lhs(5,i,j,k)
     rhs(1,i,j,k)=fac1*rhs(1,i,j,k)
     rhs(2,i,j,k)=fac1*rhs(2,i,j,k)
     ....
     lhs(4,i+1,j,k)=fac2*lhs(4,i+1,j,k)
     lhs(5,i+1,j,k)=fac2*lhs(5,i+1,j,k)
     rhs(1,i+1,j,k)=fac2*rhs(1,i+1,j,k)
     rhs(2,i+1,j,k)=fac2*rhs(2,i+1,j,k)
     ....
```

yloop-7: natural loop order, rolled-up i-loop

```
  do 2  k=1,nz
   do 2  i=1,nx
    do 2  j=1,ny-2
     fac1        =1.d0/lhs(3,i,j,k)
     lhs(4,i,j,k)=fac1*lhs(4,i,j,k)
     lhs(5,i,j,k)=fac1*lhs(5,i,j,k)
     rhs(1,i,j,k)=fac1*rhs(1,i,j,k)
     rhs(2,i,j,k)=fac1*rhs(2,i,j,k)
     ....
```

A.2: Scalar window codes.

Only optimization *z-7* (natural loop order) is shown. The code fragment employs a window of scalars that holds three rows of the pentadiagonal matrix and the corresponding three triplets of right hand side values.

zloop-7: natural loop order, rolled-up i-loop

```
  do 2  j=1,ny
   do 2  i=1,nx
    !initialize scalar window (3 matrix rows)
    lhs3 = lhs(3,i,j,1)
    lhs4 = lhs(4,i,j,1)
    lhs5 = lhs(5,i,j,1)
    rhs1 = rhs(1,i,j,1)
    rhs2 = rhs(2,i,j,1)
    rhs3 = rhs(3,i,j,1)
    lhs2kp1 = lhs(2,i,j,2)
    lhs3kp1 = lhs(3,i,j,2)
    lhs4kp1 = lhs(4,i,j,2)
    lhs5kp1 = lhs(5,i,j,2)
    rhs1kp1 = rhs(1,i,j,2)
    rhs2kp1 = rhs(2,i,j,2)
    rhs3kp1 = rhs(3,i,j,2)
    lhs1kp2 = lhs(1,i,j,3)
    lhs2kp2 = lhs(2,i,j,3)
    lhs3kp2 = lhs(3,i,j,3)
    lhs4kp2 = lhs(4,i,j,3)
    lhs5kp2 = lhs(5,i,j,3)
    rhs1kp2 = rhs(1,i,j,3)
    rhs2kp2 = rhs(2,i,j,3)
    rhs3kp2 = rhs(3,i,j,3)
    !start actual iterations
    do 2  k=1, nz-2
     fac1 = 1.d0/lhs3
     lhs4 = fac1*lhs4
     lhs5 = fac1*lhs5
     rhs1 = fac1*rhs1
     rhs2 = fac1*rhs2
     rhs3 = fac1*rhs3
     lhs3kp1 = lhs3kp1-lhs2kp1*lhs4
     lhs4kp1 = lhs4kp1-lhs2kp1*lhs5
     rhs1kp1 = rhs1kp1-lhs2kp1*rhs1
     rhs2kp1 = rhs2kp1-lhs2kp1*rhs2
     rhs3kp1 = rhs3kp1-lhs2kp1*rhs3
     lhs2kp2 = lhs2kp2-lhs1kp2*lhs4
     lhs3kp2 = lhs3kp2-lhs1kp2*lhs5
     rhs1kp2 = rhs1kp2-lhs1kp2*rhs1
     rhs2kp2 = rhs2kp2-lhs1kp2*rhs2
```

```
rhs3kp2 = rhs3kp2-lhs1kp2*rhs3
!write 1 matrix row (updated elmts only)
lhs(4,i,j,k) = lhs4
lhs(5,i,j,k) = lhs5
rhs(1,i,j,k) = rhs1
rhs(2,i,j,k) = rhs2
rhs(3,i,j,k) = rhs3
!move window of scalar temporaries
lhs3 = lhs3kp1
lhs4 = lhs4kp1
lhs5 = lhs5kp1
rhs1 = rhs1kp1
rhs2 = rhs2kp1
rhs3 = rhs3kp1
lhs2kp1 = lhs2kp2
lhs3kp1 = lhs3kp2
      lhs4kp1 = lhs4kp2
      lhs5kp1 = lhs5kp2
      rhs1kp1 = rhs1kp2
      rhs2kp1 = rhs2kp2
      rhs3kp1 = rhs3kp2
      !read new matrix row
      lhs1kp2 = lhs(1,i,j,k+3)
      lhs2kp2 = lhs(2,i,j,k+3)
      lhs3kp2 = lhs(3,i,j,k+3)
      lhs4kp2 = lhs(4,i,j,k+3)
      lhs5kp2 = lhs(5,i,j,k+3)
      rhs1kp2 = rhs(1,i,j,k+3)
      rhs2kp2 = rhs(2,i,j,k+3)
      rhs3kp2 = rhs(3,i,j,k+3)
   2 continue
```

Appendix B: Measured computational performance (Mflops/s).

B.1: Kernel code performance.

Table 6: MIPS R5000

	grid size			
	16	32	64	80
x-1	8.34 •	8.70 •	8.86 •	8.71 •
x-2	8.73 •	9.10 •	9.25 •	9.09 •
x-3	14.0●	15.2●	15.8●	16.3●
x-4	17.2●	18.3●	18.9●	19.0●
x-5	17.5●	18.7●	19.3●	19.5●
y-1	7.95 •	8.20 •	8.30 •	7.05 •
y-2	8.09 •	8.34 •	8.42 •	7.15 •
y-3	11.8●	12.5●	12.8●	9.87 •
y-4	15.0●	16.0●	16.5●	16.5●
y-5	15.7●	17.0●	17.6●	17.7●
y-6	15.8●	17.1●	16.3●	16.6●
y-7	15.2●	16.4●	14.9●	15.6●
z-1	6.69 •	6.12 •	6.04 •	6.00 •
z-2	6.71 •	6.14 •	6.05 •	6.01 •
z-3	9.76●	8.37●	8.46●	8.29●
z-4	7.54•	7.04•	6.90•	7.01•
z-5	7.58•	7.05•	7.10•	7.02•
z-6	13.2●	12.7●	12.2●	11.6●
z-7	12.4●	11.1●	10.7●	10.1●

Table 7: MIPS R8000

	grid size			
	16	32	64	80
x-1	23.0•	23.2•	17.8•	17.8•
x-2	23.0•	23.2•	17.8•	17.8•
x-3	51.8●	53.9●	29.5●	29.5●
x-4	48.5●	49.7●	28.6●	28.5●
x-5	46.3●	47.0●	27.4●	27.4●
y-1	49.8•	62.5•	30.0•	30.4•
y-2	49.8•	62.6•	30.1•	30.5•
y-3	83.1●	95.6●	38.9●	38.8●
y-4	82.7●	95.1●	39.1●	39.2●
y-5	28.5 ·	29.0 ·	20.6•	20.6•
y-6	28.5 ·	29.0 ·	20.6•	20.6•
y-7	82.7●	95.1●	39.1●	39.2●
z-1	29.3 ·	31.4 ·	22.1•	20.9•
z-2	29.3 ·	31.4 ·	22.1•	20.9•
z-3	83.5●	95.9●	37.8●	36.5●
z-4	82.9●	95.3●	39.1●	39.0●
z-5	28.6 ·	29.0 ·	20.6•	20.5•
z-6	28.6 ·	29.0 ·	20.6•	20.5•
z-7	82.9●	95.3●	39.1●	39.0●

Table 8: MIPS R10000

	grid size			
	16	32	64	80
x-1	55.9•	59.7•	49.6●	48.8●
x-2	55.8•	59.7•	49.6●	48.7●
x-3	86.1●	89.5●	70.8●	68.2●
x-4	127.●	131.●	60.1●	60.6●
x-5	116.●	116.●	63.4●	63.9●
y-1	85.4●	88.0●	63.4●	61.9●
y-2	85.3●	88.0●	63.5●	61.7●
y-3	92.2●	97.2●	81.0●	61.4●
y-4	107.●	113.●	69.0●	61.2●
y-5	89.7●	91.5●	52.2•	52.4●
y-6	89.7●	90.4●	52.2•	52.4●
y-7	108.●	114.●	69.1●	61.8●
z-1	77.2●	72.9●	50.0●	51.5●
z-2	77.9●	73.0●	50.0●	51.7●
z-3	80.3●	72.1●	63.5●	60.3●
z-4	104.●	103.●	59.1●	51.5●
z-5	90.6●	89.6●	49.8●	41.4●
z-6	90.2●	89.6●	49.7●	41.3●
z-7	105.●	103.●	59.0●	51.4●

Table 9: IBM POWER2

	grid size			
	16	32	64	80
x-1	34.3•	33.4•	33.7•	34.3•
x-2	35.0•	34.1•	34.2•	35.2•
x-3	40.5•	39.2•	39.6•	40.9•
x-4	48.9●	49.7●	50.3●	50.4●
x-5	69.0●	65.9●	67.3●	70.0●
y-1	32.8●	32.6●	33.0●	33.4●
y-2	32.0●	31.9●	32.3●	32.7●
y-3	36.4●	36.0●	36.5●	36.9●
y-4	35.5●	34.7●	35.1●	35.1●
y-5	39.4●	38.8●	39.0●	39.1●
y-6	43.0●	41.5●	42.1●	42.2●
y-7	41.0●	40.4●	40.9●	40.9●
z-1	33.7●	32.7●	29.0●	31.6●
z-2	32.6●	31.6●	28.6●	31.3●
z-3	37.4●	36.1●	31.2●	34.4●
z-4	35.6●	34.7●	29.3●	29.3●
z-5	39.6●	38.9●	31.8●	31.9●
z-6	43.8●	42.6●	42.0●	37.6●
z-7	41.9●	41.4●	40.4●	32.5●

Table 10: PentiumPro

	grid size			
	16	32	64	80
x-1	16.0●	16.0●	16.4●	16.5●
x-2	16.5●	16.5●	16.9●	17.0●
x-3	21.4●	21.4●	22.1●	22.4●
x-4	22.7●	22.6●	23.5●	23.5●
x-5	21.8●	21.8●	22.4●	22.4●
y-1	17.1●	17.1●	17.0●	17.0●
y-2	17.1●	17.1●	17.0●	17.0●
y-3	21.1●	21.1●	21.1●	21.3●
y-4	21.9●	21.9●	22.6●	22.7●
y-5	22.0●	22.0●	22.8●	22.9●
y-6	21.9●	21.9●	22.4●	22.3●
y-7	21.3●	21.3●	21.9●	21.7●
z-1	15.0●	15.0●	10.6•	10.7•
z-2	15.0●	15.0●	10.6•	10.8•
z-3	17.3●	17.3●	10.7•	10.8•
z-4	8.82•	8.82•	7.81 ·	7.64 ·
z-5	8.89•	8.89•	7.86 ·	7.69 ·
z-6	21.4●	21.4●	22.3●	22.3●
z-7	21.1●	21.1●	22.0●	22.0●

Table 11: DEC Alpha EV4

	grid size			
	16	32	64	80
x-1	7.19 ·	6.69 ·	6.81 ·	6.74 ·
x-2	7.19 ·	6.69 ·	6.81 ·	6.74 ·
x-3	9.53 •	8.50 ·	8.45 ·	8.62 ·
x-4	23.5●	23.7●	23.9●	24.0●
x-5	21.0●	21.3●	21.4●	21.4●
y-1	11.3●	9.22•	8.42•	7.42•
y-2	11.3●	9.23•	8.43•	7.43•
y-3	9.08•	7.52•	7.52•	6.92•
y-4	13.8●	12.8●	11.2●	10.7●
y-5	11.9●	11.2●	10.1•	9.78•
y-6	16.8●	16.7●	16.3●	15.7●
y-7	12.6●	12.4●	12.1●	11.7●
z-1	9.34•	7.98•	7.94•	6.90•
z-2	9.34•	7.98•	7.94•	6.90•
z-3	7.61 •	6.53 •	6.59 •	6.50 •
z-4	8.58•	8.12•	7.82•	7.78•
z-5	8.45•	8.03•	7.79•	7.75•
z-6	14.7●	14.0●	13.0●	13.2●
z-7	11.4●	10.8●	10.3●	10.4●

Table 12: DEC Alpha EV5

	grid size			
	16	32	64	80
x-1	58.3	58.2	60.0	62.2
x-2	58.4	58.1	60.0	62.2
x-3	57.3	56.1	57.0	60.4
x-4	103.	111.	116.	117.
x-5	76.6	87.9	93.9	96.1
y-1	50.0	49.4	50.9	42.2
y-2	50.2	49.4	50.9	42.2
y-3	50.0	48.7	50.4	42.1
y-4	85.0	92.0	89.9	88.0
y-5	80.0	85.6	81.6	80.1
y-6	42.8	42.3	37.1	42.8
y-7	43.5	42.1	33.9	40.7
z-1	42.1	26.4	22.5	22.5
z-2	42.1	26.3	22.5	22.5
z-3	42.0	25.7	22.2	22.3
z-4	62.8	40.7	39.3	39.5
z-5	60.3	38.1	36.1	36.2
z-6	42.5	41.7	42.6	38.5
z-7	44.4	41.6	40.8	30.4

Table 13: Sun UltraSparc

	grid size			
	16	32	64	80
x-1	31.9	24.5	25.0	25.1
x-2	33.6	25.3	25.7	25.8
x-3	32.0	24.5	25.1	25.2
x-4	45.7	31.0	31.6	31.7
x-5	42.0	29.4	30.0	30.1
y-1	28.2	24.5	25.0	22.3
y-2	28.8	24.8	25.3	22.4
y-3	28.7	24.7	25.1	21.9
y-4	32.5	24.1	23.7	23.4
y-5	34.5	25.3	25.0	24.7
y-6	34.5	25.9	26.4	26.4
y-7	33.0	24.8	25.2	25.2
z-1	28.0	21.9	19.7	19.7
z-2	28.7	22.0	19.7	19.8
z-3	27.8	21.4	19.3	19.3
z-4	27.5	20.0	16.2	15.3
z-5	28.8	20.9	17.1	16.1
z-6	34.5	21.9	19.8	19.9
z-7	33.6	19.1	15.5	15.5

Table 14: HP PA-RISC

	grid size			
	16	32	64	80
x-1	71.1	28.2	27.9	28.3
x-2	59.9	23.6	23.2	23.6
x-3	73.1	27.8	26.9	27.0
x-4	90.5	34.3	33.3	33.2
x-5	134.	35.9	34.8	35.2
y-1	57.8	23.3	22.5	22.9
y-2	58.8	23.3	22.6	23.2
y-3	74.9	28.8	27.5	28.3
y-4	86.3	30.4	30.2	30.2
y-5	117.	30.6	30.4	30.3
y-6	71.0	26.3	26.5	26.4
y-7	78.1	30.5	30.6	30.6
z-1	60.4	14.1	20.4	13.6
z-2	62.6	13.9	19.7	13.6
z-3	76.1	16.4	24.6	16.0
z-4	84.9	27.6	25.6	22.8
z-5	116.	28.2	26.1	24.0
z-6	73.6	28.4	28.6	15.9
z-7	76.1	30.8	31.3	10.7

Table 15: Cray J90

	grid size			
	16	32	64	80
x-1	44.3	63.6	79.8	68.3
x-2	44.3	63.0	75.0	66.9
x-3	43.7	65.0	77.7	68.1
x-4	41.9	61.6	74.9	67.6
x-5	26.3	43.0	57.5	64.2
y-1	31.7	49.5	64.4	55.6
y-2	31.9	49.2	65.0	54.8
y-3	33.0	50.0	65.4	53.7
y-4	43.8	61.9	79.4	67.7
y-5	27.2	43.0	58.2	65.1
y-6	26.3	41.6	62.2	67.3
y-7	44.5	65.1	74.7	65.0
z-1	31.4	46.7	61.5	52.4
z-2	30.8	46.8	63.3	52.6
z-3	31.2	47.5	62.7	52.9
z-4	43.7	63.5	77.7	66.4
z-5	26.2	41.8	58.7	63.8
z-6	26.5	41.9	60.7	64.4
z-7	43.7	62.9	77.3	69.2

Table 16: Cray C90

	grid size			
	16	32	64	80
x-1	166.	260.	378.	386.
x-2	164.	263.	382.	394.
x-3	164.	264.	389.	386.
x-4	158.	236.	317.	325.
x-5	87.9	153.	239.	263.
y-1	95.9	172.	282.	313.
y-2	96.0	172.	283.	315.
y-3	96.1	172.	282.	312.
y-4	158.	245.	333.	344.
y-5	84.1	146.	225.	249.
y-6	85.1	148.	226.	254.
y-7	159.	246.	318.	327.
z-1	101.	180.	287.	314.
z-2	101.	179.	289.	314.
z-3	101.	180.	288.	316.
z-4	158.	247.	330.	340.
z-5	84.2	146.	226.	252.
z-6	85.6	147.	228.	253.
z-7	159.	243.	323.	341.

B.2: Scalar window code performance

Only natural loop orderings and unrolled `m`-loops are amenable to the scalar window optimizations. Hence, only optimizations *xloop-3,4,5*, *yloop-6,7*, and *zloop-6,7* are presented.

Table 17: MIPS R5000

grid size

	16	32	64	80
x-3	13.6	14.8	14.9	15.1
x-4	13.6	14.5	15.0	15.1
x-5	12.9	13.7	14.2	14.3
y-6	12.6	13.5	13.0	13.3
y-7	13.0	14.0	12.8	13.5
z-6	12.4	12.0	11.8	11.6
z-7	12.8	11.7	11.2	11.0

Table 18: MIPS R8000

grid size

	16	32	64	80
x-3	62.8	66.2	35.7	36.6
x-4	62.9	66.2	36.9	36.9
x-5	52.5	54.7	28.2	28.0
y-6	51.7	53.9	27.8	27.7
y-7	57.1	62.0	35.7	36.2
z-6	51.2	53.3	27.8	25.0
z-7	57.4	62.2	36.1	30.1

Table 19: MIPS R10000

grid size

	16	32	64	80
x-3	139.	147.	95.3	93.8
x-4	139.	146.	67.1	67.8
x-5	131.	142.	67.2	67.8
y-6	126.	135.	73.2	64.0
y-7	134.	140.	73.9	64.6
z-6	119.	100.	28.0	26.7
z-7	127.	103.	28.3	27.1

Table 20: IBM POWER2

grid size

	16	32	64	80
x-3	64.5	62.9	64.3	65.6
x-4	66.4	66.0	66.0	66.0
x-5	54.0	51.8	52.3	53.1
y-6	49.7	47.3	47.7	47.7
y-7	58.0	56.9	58.4	58.3
z-6	50.8	48.6	47.4	41.2
z-7	60.9	60.3	58.2	43.2

Table 21: PentiumPro

grid size

	16	32	64	80
x-3	31.7	23.4	24.3	24.6
x-4	33.2	22.6	23.3	23.5
x-5	33.2	22.7	23.4	23.5
y-6	28.5	20.8	21.7	21.8
y-7	26.4	19.6	20.3	20.3
z-6	28.4	20.8	21.6	21.8
z-7	28.0	20.4	20.9	20.9

Table 22: DEC Alpha EV4

grid size

	16	32	64	80
x-3	11.1	11.4	11.6	11.6
x-4	12.7	13.1	13.3	13.4
x-5	11.0	11.4	11.5	11.6
y-6	11.2	11.4	11.3	11.1
y-7	12.9	13.3	12.9	12.8
z-6	10.9	11.1	11.1	11.2
z-7	13.6	13.4	14.2	13.8

Table 23: DEC Alpha EV5

grid size

	16	32	64	80
x-3	76.9	74.6	79.3	80.8
x-4	115.	117.	121.	122.
x-5	51.4	56.5	59.7	60.3
y-6	35.6	35.0	33.5	35.6
y-7	53.9	53.5	42.0	52.3
z-6	34.9	34.5	35.4	31.9
z-7	55.5	53.4	52.8	41.1

Table 24: Sun UltraSparc

grid size

	16	32	64	80
x-3	45.4	31.2	32.2	32.5
x-4	44.8	31.2	32.1	32.3
x-5	35.4	26.7	27.5	27.7
y-6	34.3	26.0	26.6	26.7
y-7	43.2	30.4	31.0	31.2
z-6	34.4	21.5	19.9	20.0
z-7	43.1	20.4	17.1	17.1

Table 25: HP PA-RISC

grid size

	16	32	64	80
x-3	100.	53.6	56.6	56.4
x-4	99.4	54.4	56.8	57.0
x-5	126.	43.7	44.6	44.7
y-6	118.	38.4	39.7	39.9
y-7	82.6	37.9	39.8	40.0
z-6	118.	40.9	41.8	19.6
z-7	95.8	50.4	54.7	12.4

Table 26: Cray J90

grid size

	16	32	64	80
x-3	21.4	31.1	41.8	35.2
x-4	20.6	30.4	41.6	34.9
x-5	12.5	20.3	30.3	34.1
y-6	12.6	20.7	30.3	34.4
y-7	20.7	30.6	42.1	34.8
z-6	12.9	21.6	32.0	35.1
z-7	21.2	31.0	42.2	35.5

Table 27: Cray C90

grid size

	16	32	64	80
x-3	64.2	109.	155.	164.
x-4	67.2	112.	158.	166.
x-5	40.3	73.0	118.	132.
y-6	40.1	72.8	116.	131.
y-7	63.7	106.	150.	161.
z-6	40.0	73.0	116.	131.
z-7	64.0	107.	150.	162.

Appendix C: Simulated cache performance (% load misses).

Table 28: MIPS R5000

	grid size			
	16	32	64	80
x-3	6.78⊙	6.12⊙	5.83⊙	5.77⊙
x-4	6.45⊙	5.97⊙	5.76⊙	5.71⊙
y-3	6.48⊙	5.98⊙	5.76⊙	9.89⊚
y-4	6.63⊙	6.06⊙	5.80⊙	5.75⊙
y-7	6.63⊙	6.06⊙	6.88⊚	6.18⊙
z-3	10.2⊚	12.9⊚	12.7⊚	12.7⊚
z-4	15.6•	16.9•	16.8•	16.8•
z-7	6.63⊙	6.06⊙	5.80⊙	5.75⊙

Table 29: MIPS R8000

	grid size			
	16	32	64	80
x-3	0.00⊙	0.00⊙	.369⊙	.369⊙
x-4	0.00⊙	0.00⊙	.369⊙	.359⊙
y-3	0.00⊙	0.00⊙	.369⊙	.359⊙
y-4	0.00⊙	0.00⊙	.369⊙	.359⊙
y-7	0.00⊙	0.00⊙	.369⊙	.359⊙
z-3	0.00⊙	0.00⊙	.369⊙	.359⊙
z-4	0.00⊙	0.00⊙	.369⊙	.359⊙
z-7	0.00⊙	0.00⊙	.369⊙	.359⊙

Table 30: MIPS R10000

	grid size			
	16	32	64	80
x-3	0.00⊙	.060⊙	1.46⊙	1.45⊙
x-4	0.00⊙	.229·	1.46⊙	1.44⊙
y-3	0.00⊙	.050⊙	1.45⊙	1.44⊙
y-4	0.00⊙	.229·	1.46⊙	1.43⊙
y-7	0.00⊙	.229·	1.46⊙	1.44⊙
z-3	0.00⊙	.060⊙	1.45⊙	1.43⊙
z-4	0.00⊙	.229·	1.46⊙	1.44⊙
z-7	0.00⊙	.110⊚	1.46⊙	1.44⊙

Table 31: IBM POWER2

	grid size			
	16	32	64	80
x-3	.636⊙	.784⊙	.735⊙	.725⊙
x-4	.587⊙	.764⊙	.725⊙	.725⊙
y-3	.537⊙	.764⊙	.725⊙	.715⊙
y-4	.587⊙	.764⊙	.725⊙	.725⊙
y-7	.547⊙	.764⊙	.754⊙	.784⊙
z-3	.507⊙	.764⊙	1.59⊚	1.59⊚
z-4	.557⊙	.764⊙	2.12•	2.11•
z-7	.587⊙	.764⊙	.725⊙	.725⊙

Table 32: PentiumPro

	grid size			
	16	32	64	80
x-3	3.98⊚	6.12⊙	5.83⊙	5.77⊙
x-4	3.22⊙	5.97⊙	5.76⊙	5.71⊙
y-3	2.84⊙	5.98⊙	5.76⊙	5.71⊙
y-4	3.72⊚	6.06⊙	5.80⊙	5.75⊙
y-7	3.73⊚	6.06⊙	5.80⊙	5.75⊙
z-3	3.18⊙	5.98⊙	12.7⊚	12.7⊚
z-4	3.54⊙	6.06⊙	16.8•	16.8•
z-7	3.74⊙	6.06⊙	5.80⊙	5.75⊙

Table 33: DEC Alpha EV4

	grid size			
	16	32	64	80
x-3	6.78⊙	6.12⊙	5.83⊙	5.77⊙
x-4	6.86⊙	6.36⊙	6.15⊙	6.11⊙
y-3	10.6⊚	5.98⊙	9.93⊚	12.7⊚
y-4	8.94⊙	10.3⊚	14.0⊚	15.2⊚
y-7	7.56⊙	7.46⊙	8.62⊙	8.98⊙
z-3	13.4⊚	12.9⊚	12.7⊚	12.7⊚
z-4	17.6⊚	17.3⊚	17.2⊚	17.2⊚
z-7	7.56⊙	8.15⊙	8.36⊙	8.36⊙

Table 34: DEC Alpha EV5

	grid size			
	16	32	64	80
x-3	6.78⊙	6.12⊙	5.83⊙	5.77⊙
x-4	6.45⊙	5.97⊙	5.76⊙	5.71⊙
y-3	6.48⊙	5.98⊙	5.76⊙	5.71⊙
y-4	6.63⊙	6.06⊙	5.80⊙	5.75⊙
y-7	6.63⊙	6.06⊙	5.80⊙	5.75⊙
z-3	6.48⊙	12.7⊚	12.7⊚	12.7⊚
z-4	6.63⊙	16.9•	16.8•	16.8•
z-7	6.63⊙	6.06⊙	5.80⊙	5.75⊙

Table 35: Sun UltraSparc

	grid size			
	16	32	64	80
x-3	0.00⊙	3.07⊙	2.91⊙	2.88⊙
x-4	0.00⊙	3.04⊙	2.91⊙	2.88⊙
y-3	0.00⊙	3.00⊙	2.88⊙	2.86⊙
y-4	0.00⊙	3.10⊙	2.99⊙	2.98·
y-7	0.00⊙	3.10⊙	2.99⊙	2.98⊙
z-3	0.00⊙	3.00⊙	4.93•	5.61⊚
z-4	0.00⊙	4.11⊙	6.89•	8.42•
z-7	0.00⊙	3.10⊙	2.98⊚	2.97⊙

Table 36: HP PA-RISC

	grid size			
	16	32	64	80
x-3	7.27⊙	6.52⊙	6.19⊙	6.12⊙
x-4	7.01⊙	6.47⊙	6.22⊙	6.17⊙
y-3	11.9⊚	6.36⊙	6.11⊙	13.6⊚
y-4	7.65⊙	7.68⊙	8.52⊙	9.03⊙
y-7	7.38⊙	6.85⊙	8.40⊙	6.89⊙
z-3	11.7⊚	3.00⊙	14.6⊚	14.5⊚
z-4	16.5⊚	4.11⊙	20.3•	20.3•
z-7	7.38⊙	3.10⊙	6.81⊙	6.79⊙